Mobile Rendering: Algorithms, Implementation, and Performance

An Honors Thesis (HONR 499)

by

Jessica Lohse

Thesis Advisor
Dr. Shaoen Wu

Signed

Ball State University
Muncie, IN

April 2016

Expected Date of Graduation
May 2016

ABSTRACT

Over the last fifty years, three-dimensional computer graphics has revolutionized the film industry. Three-dimensional images on mobile devices, however, have not followed suit because of the small amount of processing power available. In this paper, a brief look into the history of algorithms that have been instrumental throughout the development of three-dimensional images is given. The findings found from creating a custom ray tracer and the differences between rendering times on a laptop versus two different mobile devices are then presented. Lastly, a summary of all of the findings and ideas about how these algorithms can be improved to run more efficiently on mobile devices are offered.

ACKNOWLEDGMENTS

A huge thanks to everyone who has helped me during my year of research, whether it was just emotional or academic. Special thanks to Dr. Shaoen Wu for agreeing to be my mentor through this process and helping me work through some problems I could not figure out myself. Another big thanks goes to Julian Fong, a developer currently working a Pixar. He took the time out of his schedule to answer some of my questions and help me understand some of the finer details of rendering. Thanks also to Blake Hair who was always there for me to bounce ideas off of, to help me understand complex math with the stuffed animals in my room, and never let me give up on what I am passionate about.

Mobile Rendering: Algorithms, Implementation, and Performance

Jessica Lohse
Ball State University

1. INTRODUCTION

Animation and the rendering required to produce 2D images from 3D objects has attracted a number of researchers interested in the attractive field since its inception. To obtain a better understanding of the rendering process completed by large animation companies, research was conducted into the different algorithms and rendering approaches that are used to create films such as 'Toy Story' and 'Inside Out.' Seven algorithms and methods that have lead us to photorealistic graphics since the early 1950s are explained in this paper: hidden-surface and line algorithms, z-buffering, scanline rendering, ray tracing, radiosity, Monte Carlo rendering, and the rendering equation.

With this knowledge and the help of both the first and second editions of Peter Shirley's *Realistic Ray Tracing*, a simple ray tracer was created. Images created with the ray tracer could include spheres, triangles, lights, simple textures of one color, generated textures, and texture mapping. In an attempt to diagnose the major pressure points of the ray tracer, images were generated that would test different systems to determine the breaking points of rendering. By looking at six different categories of image tests, some guidelines for rendering on mobile devices and some suggestions on how to improve rendering times were developed.

2. HISTORY OF COMPUTER GRAPHICS

Before computers were commercially available, government agencies and government-funded universities started the field of computer graphics. The Whirlwind computer, developed at the Massachusetts Institute of Technology in the early 1950s, was the first computer with a CRT screen that showed the solutions to differential equations. The U.S. Air Force used CRT screens to display aircrafts that were flying over the U.S. in the mid- to late-1950s. Ivan Sutherland created the first interactive system in 1962 at MIT [5]. Called Sketchpad, the system implemented the beginnings of one of the first algorithms that have become essential to three-dimensional computer graphics, the hidden-surface algorithm.

2.1 Hidden-Surface and Line Algorithms

Sutherland, Robert F. Sproull, and Robert A. Schumacker did an intensive study where they characterized all hidden-surface and –line algorithms into three different classes [6]. The first class is made up of object-space algorithms. Object-space algorithms focus on rendering the image "exactly" as it should be, looking the same even after being enlarged. These algorithms ask if each item in the environment is visible or not. As the complexity of the environment grows, so does the performance costs for these algorithms. Object-space algorithms render their images in three-dimensional space, which allows for the generation of data that can be used to improve the rendering of textures, shadows, and anti-aliasing, or the removal of jagged edges to an object [5].

The second class identified by the three men was the image space algorithms. These algorithms are performed usually at the resolution of the display screen presenting the image. Instead of asking if each object is visible to the viewer like in object-space, image-space algorithms ask whether an object is visible to a certain dot on the screen, projecting each model from a three-dimensional environment to a two-dimensional plane. The cost of these algorithms is constant, seeing as the resolution of a screen will never change [6]. While being more efficient, the three-dimensional information of objects is lost, which will cause the rendering of shadows, textures, and anti-aliasing enhancement to be difficult [5].

Lastly, list-priority hidden-surface algorithms are a mixture of both object-space and image-space algorithms. Like image-space algorithms, images are created for a fixed-resolution display. However, image-space algorithms calculate at what depth the viewing ray intersects the surface and then determines if the surface is visible. List-priority algorithms compute the visibility in object-space first and then render the image in image-space. The object-space calculations allow for high precision depth calculations to be performed. While computing using this method may be intensive, the same list can be used to generate many frames in an animation if the scene does not change very often [6].

2.2 Z-Buffering

To solve the problem of lost information for shadows in image-space algorithms, Edwin Catmull introduced the idea of the z-buffer in his thesis at University of Utah [3]. When the image-space algorithm used determines the z value and intensity of an object, both values are stored for every pixel on the image. If another object is also placed at that same pixel, the z value of the new object is compared to the currently stored z value. If the new z value is larger, the new object is behind the other object and nothing is written to the z-buffer. Otherwise, the new z value will replace the z value currently in the buffer.

The z-buffer solves the intersections of surfaces and hidden surface problems mentioned in the image-space section above rather trivially. It also allows images to be of any complexity and have as many objects as an artist wanted, however, extra memory is needed to store the values and there must be enough precision in z values to determine which object is in front of another.

Z-buffering introduces two interesting anti-aliasing problem at the same time. If two objects have the same z value, the incorrect z value may be stored in the buffer, thus causing unwanted artifacts or incorrect pixels to be rendered on the image [9]. The intensity at each pixel on the silhouette of an object is the combination of the intensities of the object displayed at that pixel and the object partially obscured. Since the z-buffer approach allows objects to be in any order, this may cause some problems when trying to

render silhouettes. To correctly avoid this problem of anti-aliasing, it may be necessary to sort the objects before using the z-buffer approach.

2.3 Scanline Rendering

Developed around the same time as Catmull's z-buffer, scanline rendering is an example of an image-space algorithm that simplifies the rendering process. This process takes an image one scanline, or row of pixels, at a time consecutively from top to bottom. Edges of polygons or objects of an image are held in a single list, which needs to be reordered when moving from one scanline to the next in case there are any overlapping edges in the next line. Each scanline can be broken into "simpler" parts to reduce the complexity on the hardware that is rendering the image by placing each part on its own rendering thread.

This rendering method has been used quite commonly in real-time rendering systems like video games or web-based applications, only recently being replaced by ray tracing as it becomes more technologically feasible. It is an efficient algorithm for generating supersampled images with low overheads, creating images and transmitting them line-by-line, and can function as input to many common image compression methods. Depending on the memory available, this algorithm may be ineffective or produce incorrect results since it requires all objects to be held in memory during the whole process [4].

2.3.1 Scanline Object Ordering

The ordering of objects is very critical to this method, seeing as the list of objects changes order for every scanline. One algorithm to sort is the painter's algorithm, which orders objects back to front but does not take into account overlapping objects and requires sorting by depth. A modified painter's algorithm places objects in a tree data structure to account of overlapping objects [9]. There are three major tree structures used in scanline rendering: binary space partitioning trees, kd-trees, and octrees.

Binary Space Partitioning Trees, or BSPs, start off with a box that encompasses the entire image scene and a determined threshold. If the number of objects in the box is greater than the threshold, the box is split in half. The process continues until every box contains fewer objects than the threshold amount or a maximum depth has been reached. Each box is then made up of all of the objects that it contains, allowing for an object to be in multiple boxes. Each box will be differently sized from the others, allowing the algorithm to handle uneven distribution of objects

Kd-trees improve the scanline rendering algorithm by making different choices on how the space is divided. Instead of splitting a box in half in whatever direction as is possible with a BSP, subdivisions are only allowed to go in one direction, perpendicular to either the x- or y-axis. Traversal and construction of this type of tree is more efficient and thus easier to render the image.

Another variation of the BSP is the octree. In this process, the image scene is split evenly by all three coordinate axes, creating eight boxes at each step. This reduces the number of times the scene needs to be split so that a maximum depth will not be reached as quickly as subdividing once [8].

The z buffer can also be used to sort through the objects. The z value of each object in a scanline is stored in what has been called an s buffer. By sorting the objects this way, the correct object for each pixel will be chosen to be rendered and less memory is used because it only stores the information for each scanline instead of the entire image [4].

2.4 Ray Tracing

Up until this point, images were not very realistic even if reflections, shadows, and transparent materials were rendered. To increase the realism in rendered images, Turner Whitted developed an algorithm that would accurately render scenes by using global illumination information for the intensity of a color at each pixel [14]. Ray tracing is viewed as one of the most advanced rendering methods because it calculates every ray of light in a scene, though this also makes it very computationally expensive [5].

The ray tracing algorithm can be demonstrated by Figure 1. The point *e* is the eye looking into the scene, the camera in many rendering situations. It looks into the 3D scene, here containing two spheres. To create the 2D image a ray is sent out from the camera to every pixel in the image plane. As the ray passes through the image plane, it enters the 3D scene and whatever object the ray hits first will be the object that is displayed at that pixel [12]. If the ray never hits an object, a predefined background color is displayed for that pixel.

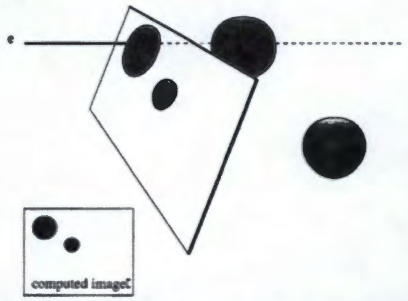

Figure 1. An Overview of the Ray Tracing Concept [12].

There are three different types of rays that are used in ray tracing. The first is a reflection ray, which is the original ray that shoots into the scene from the eye. It travels in a straight line, bouncing off reflective surfaces at the same angle that it hit the object at. Once an object has been hit and the intensity and color has been calculated for that object, shadow rays are sent from the point it intersected the object at to each light source in the scene. If the shadow ray hits another object before it reaches a light source it means that the object it originated from is not visible to the light source and will be in the shadow of other objects. If a ray encounters a transparent surface, a refraction ray is generated to calculate the amount of light refraction that will pass through the object and at what angle, depending on the thickness of the surface of the transparent object.

Since ray tracing calculates an image in 3D space, it is very precise about simulating the behavior of light [5]. It also makes displaying mirror reflections and refractions, direct illumination, detailed textures, shiny surfaces, and shadows very accurate to what we would see in real life. Curved surfaces are also very simple to render using this method, seeing as a ray can hit the surface at any point and calculate its color. The fact that ray tracers calculate each pixel independently allows for this method to easily be parallelized and cut down on computation costs and time [14]. It is also possible to combine ray tracing with other algorithms. In Pixar's PRMan renderer, objects directly visible to the camera are rendered with their own scanline algorithm,

REYES. Once those objects have been shaded, those pixels can be used to create rays for ray tracing, completely removing all reflection rays from the computations.

Ray tracing works well and is efficient only if the entire scene can fit into memory, otherwise the renderer will need to recreate objects [10]. Similarly, as the complexity of the scene grows, the time needed to create the scene grows. To render more detailed images with a higher pixel count takes longer because each increase in pixels is an increase in rays and calculations that must occur. The same can be said once more light sources are added to the scene, since more shadow rays will need to be generated [5]. The number of rays grows exponentially with each bounce off of an object and it is possible that each ray could hit a new object, both of these causing the algorithms to be inefficient [9, 14]. This method only accounts for point and directional light sources, causing sharp edged shadows to be rendered. Ray tracing also does not account for indirect illumination, though adding an ambient light can fix this problem [5].

2.5 Radiosity

Four scientists from Cornell University took findings from thermal engineering methods to produce the radiosity method for computer graphics The radiosity method accurately shows the diffusion of light between objects and computes global illumination effects accurately, removing the need to add ambient lighting from ray tracing algorithms [2]. The theory behind the method is that every object is a light source in its own right. When light hits an object, the light is absorbed into the object and then part of that light is then reflected onto other objects [14].

To determine the amount of reflected light an object gives off, the scene is divided into small sections. The algorithm then models, for each of these patches, the light transfer between objects. Each pixel in a particular patch has the same light result, causing the image to look like a patchwork quilt [5, 9]. Once each patch has been calculated, additional processing, known as radiosity reconstruction, must take place.

There are three different methods for radiosity reconstruction. The first interpolates and smoothes out the values between patches, making each patch blend into the other nicely. Another is to take the exiting radiance value from each patch as a ray and then take a sampling from each ray at the different directions to determine the final light value. The usage of ray traced specular highlights and mirror reflections can also factor into the radiosity reconstruction to add more realism to the image [9].

The type, distance, and positioning of each object is very important in radiosity. For the method to work, all objects have to have diffuse surfaces. Diffuse surfaces, or Lambertian surfaces, have one color, no matter where the viewer is looking at the object from, such as the surface of cardboard [2]. Calculating the reflections from polished marble floors and large mirrors will not work using this method. The distance between each object determines how much of the light will be transferred to the object. Objects that are parallel to each other will pass more light onto each other than those that are perpendicular since the light rays will more easily bounce between them [5].

The radiosity method gives a solution for a static scene. Once more lights are added or objects are moved, the solution is invalidated [5]. This method is also very memory intensive and time consuming. Objects with lots of details require large amounts of subdividing, causing an explosion in processing time and memory needed [9].

2.6 Monte Carlo Rendering

The addition of radiosity methods still did not give rendered images enough realism to produce photorealistic images. While objects were now lit more accurately, ray tracing only renders sharp shadows, reflections, and refractions. A group of three scientists in the Computer Division at Lucasfilm Ltd. developed the approach of distributed ray tracing to achieve the look of motion blur, depth of field, translucency, and fuzzy reflections in 1984 [13].

The scientists used the Monte Carlo method as an inspiration for their rendering algorithm and it has since been called Monte Carlo Rendering. The Monte Carlo method returns different results based on the randomized values it uses. When it is too hard or complex to find an answer analytically, taking the average of the results of several runs of this algorithm will produce an answer that is statistically very likely the answer [8, 9]. Instead of adding more rays to a scene to determine what an object looks like at a pixel, rays are distributed in time and are sent from at different locations to oversample the image. Taking an average of these values will give the correct material for that pixel that is no longer as harsh as when just using ray tracing.

Sampling at different places and times in a scene in an image create each effect. Distributing rays throughout time and averaging their result creates motion blur. Producing rays at different distances from the object gives the image depth of field. Sampling of the reflection ray produces glossy or blurred reflections, while sampling the refraction ray produces blurred transparency or translucency [13].

Because many calculations that are heavily used in computer graphics are complex integrals, Monte Carlo rendering allows for a relatively quick and inexpensive way to get a close estimation. This algorithm only needs to evaluate an integrand at arbitrary points to give an estimate. Thus Monte Carlo rendering is easy to implement and allows for solving various types of integrands, including ones that contain discontinuities.

For Monte Carlo rendering to predict an accurate result more samples are needed. Too few samples will result in not enough paths to be representative of the image. The difference between the actual result and the estimation manifests itself as noise throughout the image [9]. To decrease the error amount between the actual result and the estimation by one half, four times as many samples are needed. These samples require one or more rays each, leading to some computationally expensive calculations to be done [8]. To decrease the amount of sampling in an image, more rays can be focused on light source that have been deemed more important, but this will lead to biases during rendering [9].

2.7 The Rendering Equation

The biggest advantage of Monte Carlo rendering is it solves the Rendering Equation given enough time. Before this equation, a fully implemented algorithm for correctly displaying global illumination did not exist. Objects could now have blurs, transparency, soft shadows, and radiosity lighting, but it was not combined with the findings of radiosity to produce diffuse lighting. James Kajiya developed the rendering equation that generalizes these algorithms together for ease of use in computer graphics [7].

The equation is as follows:

$$L_O(x,\vec{w}) = L_e(x,\vec{w}) + L_r(x,\vec{w}) \quad (1)$$

$$= L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w})L_i(x,\vec{w}')(\vec{n}\cdot\vec{w}')d\vec{w}' \quad (2)$$

L_O is the radiance of light coming from an object in a particular direction, $\vec{w}$, at a pixel, x. The radiance is sum of the light emitted from a surface, L_e, and the light reflected from all incoming directions, L_r, including light that is emitted from reflective surfaces. Equation 2 further expands L_r into the integral of the incoming radiance over the hemisphere of directions above the surface point at a surface normal of $\vec{n}$. In this equation, f_r is the function that determines how light is reflected onto an opaque surface and L_i is the incoming light at x, which may either be light directly from a light source or reflected from another surface [9]. Ω represents all of the surfaces of all of the objects in the scene [7].

In other words, the rendering equation calculates the amount of light reaching the camera from any point on an object, given the sum of all the light emitted or reflected from the object [8]. It uses both aspects of radiosity and ray tracing to efficiently compute the combined effect of all of the various sets of paths in the scene [11]. The Monte Carlo method allows for this equation to be estimated to a sufficiently accurate approximation to produce photorealistic images. Actually solving this equation should only occur for the simplest of scenes [8].

2.8 Since Then

With the introduction of the rendering equation, photorealistic images could be computed and rendered. Since its creation scientists have been working on trying to optimize not only the equation itself but also the other parts of rendering that have been explained above. The goal is to make rendering 3D scenes as quick and efficient as possible.

This brief summary in no way accounts for the entirety of algorithms that are used in rendering. There are also algorithms to determine particle effects such as fog and fire, adding texture to a surface so it is not flat, applying images to a surface of an object, and more. These advances, combined with the ever-growing speed and memory capabilities of our computers, have allowed for our renderings to become more realistic and be produced at faster speeds. There is still much work to be done, especially since consumers are quickly using their mobile devices to generate images that have nowhere near the computing power that the specialized rendering servers animation companies use have.

3. IMPLEMENTATION FOR MOBILE DEVICES

To develop some theories for what should be tweaked for mobile devices, it is first needed to find a way to render 3D images. Instead of using already existing programs or using files from 3D software such as Maya, a custom ray tracer was built. By doing so control over what the product would be was gained. It would also allow for implementation of any of the rendering algorithms described above that was determined to be applicable.

3.1 Building A Ray Tracer

Two of the books that were used for research on rendering algorithms gave step-by-step descriptions of how to build a ray tracer, *Physically Based Rendering* and *Realistic Ray Tracing*. *Realistic Ray Tracing*'s instructions were more concise and focused on the implementation rather than the theory of ray tracing. Both books wrote their implementations in C++, but the custom ray tracer was written in Java for two reasons. The first was that by doing so the program could easily be adapted to also run on Android mobile devices for testing. The primary researcher also was more familiar with Java than C++ so the program could quickly be developed instead of trying to learn a new programming language. About half of the book was used to develop the ray tracer until a working program that could display images with a variety of different objects, lights, and textures was made.

3.1.1 Implementation

Shirley begins by describing the necessary lower level pieces that are needed for the ray tracer. One piece is the RGB class to store the color of a pixel in the typical color format of red, green, and blue. This class includes methods to add and subtract colors, multiply and divide colors, and to turn the values of each color into byte form for use in the Image class. The Image class is made up of an array of RGB instances that designate each pixel in the image. Surfaces, lights, and cameras can be added to the image and the image will be populated with the correctly colored pixels. The array can then be converted into an actual PNG image through the use of Java's built in BufferedImage class.

The essential parts for the ray tracer were implemented next. Instances of the Vector class are used to represent both 2D and 3D points and to represent the typical vector containing a direction or offset. Instead of directly specifying the direction of a Vector, the direction is inferred from the origin point of (0,0) to the given point stored in the Vector. Methods to determine a Vector's length, the dot product, cross product, addition, subtraction, multiplication, and division by another Vector, and creating the unit Vector are all implemented. The Ray class then is made up of an origin point as a Vector and a direction as another Vector, with a single method to point the Ray at a certain point in time by adding to the origin Vector a scaled up version of the distance Vector. The Image class uses this method to determine the color of a pixel.

The next two classes created were the OrthonormalBasis and TransformMatrix classes. An instance of the OrthonormalBasis class represents the coordinate system of an object or the scene made up by three Vectors. While these may be the typical x-, y-, and z-axes (known as the canonical basis) that are typically used in mathematics, each object has its own coordinate system in relation to the canonical basis. In order to perform calculations for movement, rotation, and scaling, the program must be able to convert between the two bases of the scene and an object. The TransformMatrix class handles all of these operations.

The ray tracer by Shirley includes two shapes, a sphere and a triangle. A class for each object was created, which both included the methods to determine if a ray hits the shape at any point, to get the color, reflectance, ambient color, and texture of the shape once hit, and to determine if the shape was hit by any shadow rays. The implementation of these methods is different, accounting for the mathematics needed to determine where on the sphere a ray hit and to determine the normal of either of point on a sphere or the face of the triangle. Each shape in a scene is placed into a SurfaceList that determines if a given ray hits any of the shapes.

With these few classes and methods filled with formulas, a simple image could be created. The image in Figure 2 was created by a 500 by 500 RGB array with two shapes in the SurfaceList, a sphere placed a (250, 250, -1000) with a radius of 150 and a triangle with vertices (300, 600, -800), (0, 100, -1000), and (450, 20, -1000). This places the triangle intersecting the sphere, but the ray tracer is able to calculate where this occurs to generate the correct image.

Figure 2. The First Ray Traced Image.

The next step was to implement lighting into the ray tracer. A Light class was created that contained a light Vector and it's color. This light is added to the Image class and when the color of each pixel is determined either the background color or the color of the object with the light calculation will be returned. When deciding what color should be returned for an object, the Image class first checks to see if any shadow rays hit the object. If so, a black color is returned. Otherwise, the color of the light is multiplied by the color of the object, with red, green, and blue values multiplied by the reflectance of the object (if it has any). Figure 3 shows an example of the usage of lights and shadows. The light is being cast from the right side of the image onto two spheres. Both spheres have their left halves in shadow, and the larger sphere is in the shadow of the smaller one, leading to the right side of the larger sphere to be obstructed by shadow as well.

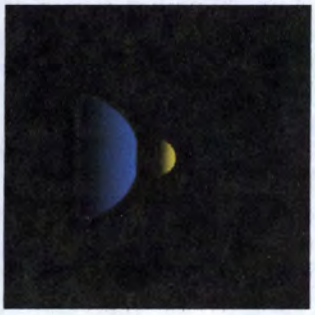

Figure 3. Shadows and Lights with Ray Tracer.

Lastly, textures were implemented so shapes did not have to have just one color to them. Three different types of textures were developed: Noise, Marble, and Image. Both the MarbleTexture and NoiseTexture use Perlin noise to randomize the pixel colors. In 1985 Ken Perlin developed Perlin noise so that textures could be generated by pseudorandom functions. Perlin noise functions are deterministic and repeatable, will never include frequency content higher than a given value, are visually random but are also continuous and smooth, and have a bounded range [1]. Using Perlin noise in my ray tracer makes generating textures simpler and takes less memory to compute. The NoiseTexture uses the getNoise function of the Perlin noise class while MarbleTexture uses the createTurbulence function. Both functions return slightly different textures each time the texture is created. Figures 4 and 5 show the differences between these two textures.

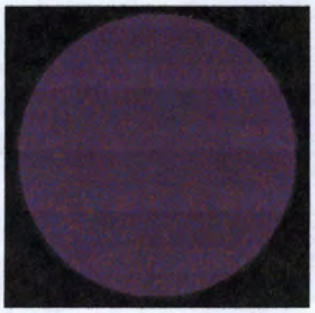

Figure 4. Generated NoiseTexture.

Figure 5. Generated MarbleTexture.

After artists have created images to represent characters or objects, those images are then wrapped around the 3D objects to give them the correct coloring and texture. An ImageTexture class was developed that allowed for a specified image to be applied to objects in a scene. A sphere was created and a flat image of the globe was picked to the texture for that sphere. The ray tracer reads through the given image, creating a new Image instance for it by reading through each pixel and storing the pixel's color. When the rendered image is being created, the Sphere class determines where on the sphere the ray has hit it and then finds the correct pixel from the input image texture. The result can be seen in Figure 7 below from the given image texture, Figure 6. For the ray tracer to correctly compute the texture, all images had to be rotated so that the ray tracer's first pixel it computed was the lower right pixel. The ImageTexture class can then work through each column until each pixel's color had been stored [11, 12].

Figure 6. Globe Image Texture.

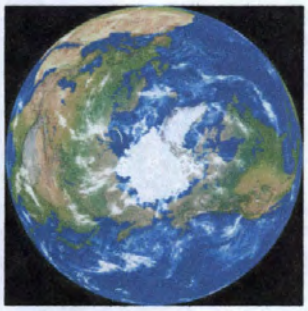

Figure 7. Generated Globe from Figure 6 Texture.

3.1.2 Creating an Android Application

Once the basic functionality of the ray tracer was finished, it was time to create an Android application so it would be possible to test the different render times across devices. Most of the program was easily adapted to be used in an Android project. There was one major problem with the work already done. The Android system does not implement the class BufferedImage, so the way images were being created and the reading in images for textures had to be modified. Instead, the Bitmap class built into Java was used to render the actual image and save the image to the phone. Since images needed to be saved locally on the phone, it was also necessary to use the FileOutputStream class to take in a file path for the image to be saved at. There may be some slight differences in rendering times because of these changes, but they are most likely very small and not worth investigating.

Figure 8 shows the layout of the application. The main page of the application shows a list of all available images that can be rendered. When the app is first opened, each cell shows a small thumbnail of what the image will look like and the name of the image. Once a user clicks on a cell, the image is created and is rendered. A pop up appears once the image has finished telling the user how long in milliseconds it took for the device to render the image. The rendering time appears on the cell for later reference and buttons allow the user to either open the file that was saved on the device or regenerate the image. These functions allowed for the ability to easily record rendering times and render an image multiple times to gather an average rendering time.

Figure 8. Android Ray Tracer Application.

3.2 Tests

The rendering times for twenty three different images were collected, split across six different categories: size of image textures, number of objects in the scene, spheres versus triangles, addition of a light to a scene, different textures, and the size of the image. Each image was rendered on three different devices with varying RAM, CPUs, GPUs, and operating systems. The specifications for each device are show in Table 1 below. For every test the image was rendered ten times and the average rendering time for each was recorded in milliseconds. These values are what are used throughout the tables in each following section.

Devices	RAM, Threads	CPU	GPU	OS, Version
MacBook Pro	16 GB DDR3, 8 Threads	Intel Core i7 2670QM	Intel Iris Pro 1536	OS X 10.11.2
HTC One M8	2 GB DDR, 4 Threads	Qualcomm Snapdragon 800 MSM8974	Adreno 330	Android 5.0.2
Samsung Galaxy S3	2 GB DDR, 2 Threads	Qualcomm Snapdragon S4 Plus MSM8960	Adreno 225	Android 4.4.2

Table 1. Testing Device Specifications

3.2.1 Image Textures

In an animation company, artists create textures for objects so they can have more complex materials with more detail than a computer could generate. The limits of using textures on mobile devices were tested for this reason. The same image was used, the flattened globe image in Figure 6, and saved at four different

sizes: 512 by 1024 pixels, 1024 by 2048 pixels, 2048 by 4097 pixels, and 4096 by 8192 pixels. Different tests were created with each size texture, applying the texture to the same sphere and seeing the results. Unfortunately, the mobile devices could only handle the smallest texture that was 512 by 1024 pixels, so it was only possible to really compare the rendering times for that image. For the larger textures, both the HTC and Samsung phone ran out of memory to store all of the pixels of the texture and render the final image.

Laptop	HTC	Samsung
639.3	12,195.6	50,919.5

Table 2. Smallest Texture Rendering Times in Milliseconds

Each device's rendering time about doubled as the device's resources decreased. The biggest factor for this is probably just the amount of threads available. The CPU information for both mobile devices is pretty similar, other than the information on threads. The HTC has double the threads and double the L2 cache. This would have allowed the HTC to render the image faster, and in the cases of larger texture images, to run out of memory quicker as it processed the pixel information faster.

The three other images were rendered using the MacBook Pro and each image took about double the amount of time to render as the previous image with a smaller texture. However, the amount of detail in the rendered image increased. The difference can be seen in Figures 9 and 10. In Figure 10, there are more colors in the clouds and the clouds are also more defined than in Figure 9. The continents have more detail as well; mountains and deserts have retained some amount of their elevation shading from the original texture. This shows how important the extra pixels are for rendering realistic objects, but rendering these images on mobile devices is difficult given the lack of immediate resources available.

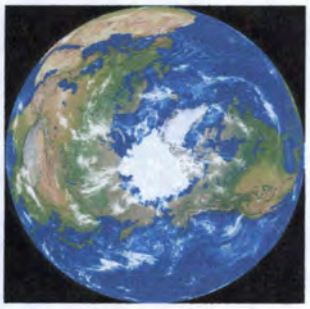

Figure 9. Generated Globe with a 512 by 1024 Pixel Texture.

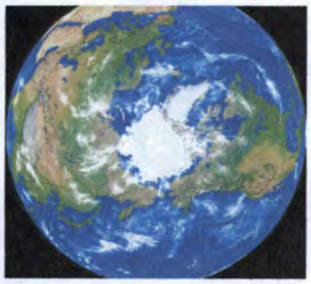

Figure 10. Generated Globe with a 4096 by 8192 Pixel Texture.

3.2.2 Number of Objects

Another big feat that animation companies have overcome with large computing devices has been how to handle crowd shots that include a large number of objects. To simulate this in smaller scale, objects were added to a scene in increments to see how additional objects increased the rendering time needed.

Figure 11 shows the progression of images that was used for testing. The first image starts off with one object, then objects are added for three objects, eleven objects, sixteen objects, twenty-one objects, and ending with twenty-four objects. The images were not completed with a set increment of objects each time, but each image shows a complete scene. Table 3 shows the rendering time for each image based on the device it was rendered on.

Table 3. Rendering Times For Objects in Milliseconds

Figure 11. The Progression of Images as New Objects were Added.

	MacBook Pro	HTC	Samsung
One	290	1,466.6	5,959.1
Three	291.8	2,180.6	9,107.9
Eleven	932.3	20,152.4	102,136.9
Sixteen	1,967.8	54,111.8	283,509.5
Twenty-One	2,974.3	93,990.4	451,019.4
Twenty-Four	3,289	115,911	518,549.2

For each platform, as more objects are added to scene the rendering time increases. The Samsung Galaxy S3 takes over 150 times more milliseconds and the HTC One took over 35 times more milliseconds than the MacBook took to render the image with twenty-four objects. This is a drastic increase and would make rendering images with these amounts of objects a little daunting to complete on mobile devices, especially if this was only one frame of an animation or game.

3.2.3 Spheres vs. Triangles

When the object tests were completed for the previous section, another trend was noticed that was causing increased rendering times. Whenever triangles were added to the scene (three to eleven objects, eleven to sixteen object, and sixteen to twenty-one objects), the rendering times needed for the image more than doubled. Tests to compare the addition of a triangle to a scene and compare it to the addition of sphere to a scene were created for this reason.

Four images were rendered: one with one triangle, one with one sphere, one with two triangles, and one with two spheres. Each sphere took up the same amount of surface area on the image as a triangle did to ensure that both objects were covering the same amount of pixels of the final image.

Tables 4 and 5 show the differences in rendering times. Depending on the device, rendering one triangle took two to seven times more milliseconds than rendering one sphere. Rendering two triangles increases this to four to ten times the amount of milliseconds to render two spheres. This can be attributed to the more complex math required to render a triangle.

Table 4. Rendering Times For One Sphere and One Triangle in Milliseconds

	MacBook Pro	HTC	Samsung
Sphere	283.5	1,768.1	7,157
Triangle	572.1	9,980.9	51,359.7

Table 5. Rendering Times For Two Spheres and Two Triangles in Milliseconds

	MacBook Pro	HTC	Samsung
Two Spheres	292.2	2,094	9,403.7
Two Triangles	840	17,030.2	91,549.8

Every triangle has three points, say a, b, and c, that make up a plane. These coordinates can be used to describe the triangle in barycentric coordinates such that:

$$p(\alpha,\beta,\gamma) = \alpha a + \beta b + \gamma c \quad (3)$$

and

$$\alpha + \beta + \gamma = 1. \quad (4)$$

For a point p to be inside the triangle, α, β, and γ all must be between zero and one by Equation 4. By solving for α, we can bring the equation down to terms of just β and γ. To determine if a ray hits any point in the plane, the equation can be expanded in x, y, and z terms and then rewritten as a standard linear equation. Using Cramer's Rule to quickly solve this 3x3 matrix equation, values for β and γ can be found. If both are between zero and one, the ray hits the triangle plane. Otherwise, it does not.

Ray sphere intersections, on the other hand, have fewer calculations. By simply solving the quadratic formula it can be determined whether the ray hits the sphere at a certain point. This explains why triangles take so much longer to render since there are multiple steps involved in deciding whether a ray has hit the triangle versus whether it has hit the sphere.

3.2.4 Lights

Since the very nature of ray tracing lends itself to take longer given more lights, two images were created: one with a light and one without. The addition of one light did not increase the amount of rendering time too dramatically. Figure 12 shows the two images and Table 6 shows the rendering times. The Samsung Galaxy still took the longest to render, which can be attributed to the lack of resources and threads as described in section 3.2.1. The closeness in rendering times may be attributed to the fact that since a light is included in the scene, before the color of the pixel is determined there must be a check to see if the shadow ray from the light also hits the pixel. If so, the pixel is given the ambient color instead of needing to calculate what that color should be, reducing the amount of color math needed to determine the pixel color. The ray tracer created did not include the ability to have more than one light, but the addition of more lights would surely increase the rendering times since there would be more rays cast from each light and other calculations needed to determine the combination of lights on the objects of the scene.

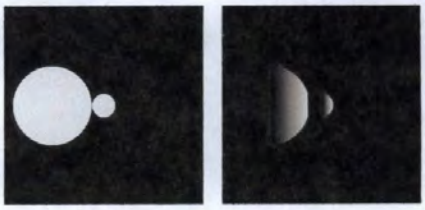

Figure 12. An Image Rendered with No Light on the Left and the Same Image Rendered with a Light Coming from the Right Side of the Objects.

Table 6. Rendering Times For an Image with a Light and One Without a Light

	MacBook Pro	HTC	Samsung
No Light	296.7	2,069.4	8,849.9
Light	321.2	2,333.1	10,258.2

3.2.5 Textures

The effects of rendering different types of textures on an object were also investigated. Rendering the same object in black and white, one color, with a noise texture, and with a marble texture served to explore this idea. As expected, the time needed to render the black and white image and the color image were basically the same.

Table 7 shows the rendering times for the noise and marble textures. For both mobile devices, it took about twice the amount of time to render the marble texture than to render the noise texture. The lack of resources on these devices causes this jump in rendering time given the large amount of computations that must be done. Just as large texture images hinder the rendering on mobile devices, generating the texture on the device dramatically slows down the rendering time.

Table 7. Rendering Times For Noise and Marble Textures in Milliseconds

	MacBook Pro	HTC	Samsung
Noise	324.5	2,218.4	9,494.4
Marble	396.3	4,855.9	21,183.8

3.2.6 Size

Lastly, the effect of different image sizes on rendering times was investigated. Images with half of the pixels taken up by a colored triangle at sizes 125 by 125 pixels, 250 by 250 pixels, and 500 by 500 pixels were created. Table 8 shows the different rendering times for each size on each device.

Table 8. Rendering Times For Different Sizes in Milliseconds

	MacBook Pro	HTC	Samsung
125 by 125 Pixels	243.5	674.4	2,826.1
250 by 250 Pixels	322.2	2,702.5	11,054.2
500 by 500 Pixels	575.2	10,947.6	52,966

Multiplying the number of pixels by four on the MacBook only increases the rendering time by about one and half times. The same quadrupling of pixels done on the two mobiles took about four to five times the rendering time needed for the smaller images. The number of threads and other resources available on the device directly attributes to this rendering time. While mobile devices can render smaller images in much shorter amount of times, there may be the need for larger images to correctly show texture complexity or for use on larger devices such as iPads or tablets.

3.2.7 Changes to The Ray Tracer for Better Rendering Times

Given a certain device, there are only so many changes that can be done to increase the rendering time. The ray tracer that was completed did not include much optimization other than some simple multiplication or use of mathematics libraries. There have been advancements that have decreased rendering times that could be implemented in the ray tracer to decrease the amount of processing power needed.

One major way to optimize the program would be to use threading to render the images. Currently, an image is rendered pixel-by-pixel, and only once a pixel's color has been determined does the program move to the next pixel. As stated in section 2, ray tracing is prime for multithreading to decrease rendering times. By splitting an image into groups and having each group rendered by a different set of threads, an image can be rendered much quicker than by just using one thread. This however could cause problems on older devices that do not have the memory or thread capabilities.

Another optimization that could be done would be to reduce the number of pixels for mobile devices. Seeing as mobile devices are generally smaller and have less of a need for high-resolution images, it could be possible to render fewer pixels and still achieve a satisfactory result. A check could be added to my ray tracer that checks the resolution of the screen on the rendering device and then adjusts the rendering algorithm for lower resolution outputs. Shooting fewer rays into the scene and reducing the number of shadow rays that get sent into the scene would also reduce the rendering time since it is not as crucial to have absolute accuracy on these smaller images.

The ray tracer was written in Java because it was the language the investigator was am most familiar with and would allow for easy testing on an Android device. However, Java does its own garbage collection of variables and caching. If this was written in C++, which is the language a majority of graphics programs are written in, the program would have better ability to decide what stays in memory and what does not. This would allow for important things to stay in the cache and free up memory for bigger calculations later on in the process. Lower level languages such as C++ also let the program define its own shader using the Shading Language and uses built in types for convenient and efficient manipulation of graphical data [1].

Precomputing colors, lights, and textures and storing the outputs could also reduce rendering times. By storing calculations that have already been computed, a simple look up can happen to determine if a result already exists for the given equation. This way less time will be spent calculating and the program can move on to other pieces of the scene. Similarly the ray tracer could determine the ambience of every object before the color calculations so less mathematics needed to be done when computing the color of the pixel.

4. CONCLUSION

Seeing as new processors and mobile devices are being released every year, eventually devices will have enough memory and resources to render complex scenes in no time. Until then, it is very important to consider how an image is rendered and what causes it to take more time than another image. Researching previous rendering algorithms gave the background knowledge needed to understand how 3D objects are changed into 2D images. Seeing the evolution of ray tracing and the various benefits it has to render images allowed the investigator to jump right in to creating semi-realistic images for multiple devices.

Through developing a ray tracer it was possible to explore some areas that could cause a drastic change in rendering times, including texture sizes, size of image, object addition, and sphere versus triangle additions. By looking at the differences it was possible to determine what would cause the increase in rendering time and possible ways to work around it. If a program could take full advantage of the parallelization available for ray tracers, determine the correct resolution needed for the rendered image, be developed in a lower level language like C++ with its own memory management, precompute and store calculations, and access enough memory to render a scene with millions of objects, ray tracing on mobile devices would be possible.

As of right now, it is only possible to pick and choose from those options. Until then, mobile games will have non-photorealistic images, most likely rendered using scanline rendering algorithms exclusively. Eventually we will be able to have realistic images rendered on mobile devices, once we have the necessary resources and ability to do so.

5. REFERENCES

[1] Anthony A. Apodaca, Larry Gritz. 1999. *Advanced Renderman: Creating CGI for Motion Picture*(1st ed.). Brian A. Barsky (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. 1984. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '84), Hank Christiansen (Ed.). ACM, New York, NY, USA, 213-222. DOI=http://dx.doi.org/10.1145/800031.808601.

[3] Edwin Earl Catmull. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. Dissertation. The University of Utah. AAI7504786.

[4] Ella Barkan, Dan Gordon. 1999. The scanline principle: efficient conversion of display algorithms into scanline mode. *The Visual Computer 15*(5), 249-264, Springer.

[5] Isaac Victor Kerlow. 2003. *The Art of 3-D Computer Animation and Effects, Third Edition* (3 ed.). M. Cummins (Ed.). John Wiley & Sons, Inc., New York, NY, USA.

[6] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. 1974. A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.* 6, 1 (March 1974), 1-55. DOI=http://dx.doi.org/10.1145/356625.356626.

[7] James T. Kajiya. 1986. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '86), David C. Evans and Russell J. Athay (Eds.). ACM, New York, NY, USA, 143-150. DOI=http://dx.doi.org/10.1145/15922.15902.

[8] Matt Pharr and Greg Humphreys. 2010. *Physically Based Rendering, Second Edition: From Theory to Implementation* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[9] Noriko Kurachi. 2011. *The Magic of Computer Graphics* (1st ed.). A. K. Peters, Ltd., Natick, MA, USA.

[10] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. "Ray Tracing for the Movie 'Cars'". Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006, pages 1-6.

[11] Peter Shirley. 2000. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA.

[12] Peter Shirley and R. Keith Morley. 2003. *Realistic Ray Tracing* (2 ed.). A. K. Peters, Ltd., Natick, MA, USA.

[13] Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '84), Hank Christiansen (Ed.). ACM, New York, NY, USA, 137-145. DOI=http://dx.doi.org/10.1145/800031.808590.

[14] Tomas Akenine-Moller, Tomas Moller, and Eric Haines. 2002. *Real-Time Rendering* (2nd ed.). A. K. Peters, Ltd., Natick, MA, USA.

```
public interface Camera {
	public Ray getRay(double x, double y, double nSubX, double nSubY);
}

public class Frame {
	OrthonormalBasis uvw;
	Vector3D origin;
	public Frame(OrthonormalBasis uvw, Vector3D origin) {
		this.uvw = uvw;
		this.origin = origin;
	}
	public OrthonormalBasis getUVW() {return uvw;}
	public Vector3D getOrigin() {return origin;}
}


public class Image {
	int rows;
	int columns;
	public RGB[][] image;
	private SurfaceList surfaces = new SurfaceList();
	private Light light;
	private Camera camera;
	private float ambience;
	public Image(int rows, int columns) {
		this.rows = rows;
		this.columns = columns;
		this.image = new RGB[columns][rows];
	}
	public void addSurface(Surface surface) {surfaces.add(surface);}
	public void addLight(Light light) {this.light = light;}
	public void addCamera(Camera camera) {this.camera = camera;}
	public void addAmbientLight(float ambience) {this.ambience = ambience;}
	public void createImage() {
		RGB[][] pixels = new RGB[this.columns][this.rows];
		for (int i = 0; i < this.rows; i++) {
			for (int j = 0; j < this.columns; j++) {
				Ray ray = this.createRay(i, j);
				if (this.surfaces.hit(ray, 0, Integer.MAX_VALUE, 0))
{
					pixels[i][j] = getHitColor(ray);
				} else {
					pixels[i][j] = getAmbientBlack();
				}
			}
		}
		populateImage(pixels);
	}
	private RGB getAmbientBlack() {
		int ambientBlack = (int) (0 + (this.ambience * 255));
		return new RGB(ambientBlack, ambientBlack, ambientBlack);
	}
	private Ray createRay(int i, int j) {
		if (this.camera == null) {
			Vector3D origin = new Vector3D(i, j, 0);
			return new Ray(origin, new Vector3D(0, 0, -1));
		} else {
			return this.camera.getRay(i, j, rows, columns);
		}
	}
	public RGB getHitColor(Ray ray) {
```

```
			Surface hitSurface = this.surfaces.getPrim();
			Vector3D hitPoint = (Vector3D) ray.pointAtParameter(this.surfaces
					.getT());
			if (this.light != null) {
				if (isHitByShadowRay(ray, hitSurface)) {
					return getAmbientBlack();
				} else {
					if (hitSurface instanceof Sphere) {
						Sphere sphere = (Sphere) hitSurface;
						return sphere.getLitColor(light, hitPoint,
ambience);
					} else if (hitSurface instanceof Triangle) {
						Triangle tri = (Triangle) hitSurface;
						return tri.getLitColor(light, ambience);
					} else {
						return getAmbientBlack();
					}
				}
			} else {
				return hitSurface.getAmbientColor(ambience, hitPoint);
			}
		}
		private boolean isHitByShadowRay(Ray ray, Surface hitSurface) {
			Vector3D originOfShadowRay = (Vector3D)
ray.pointAtParameter(hitSurface
					.getT());
			Ray shadowRay = new Ray(originOfShadowRay,
light.getLightVector());
			return this.surfaces.hit(shadowRay, 0.001, Integer.MAX_VALUE, 0);
		}
		public void populateImage(RGB[][] pixels) {
			image = pixels;
		}
		public void printImage(String imageName) throws IOException {
			BufferedImage img = new BufferedImage(columns, rows,
					BufferedImage.TYPE_INT_RGB);
			for (int i = 0; i < rows; i++) {
				for (int j = 0; j < columns; j++) {
					RGB rgb = image[i][j];
					int color = (rgb.red << 16) | (rgb.green << 8) |
rgb.blue;
					img.setRGB(j, i, color);
				}
			}
			File file;
			if (imageName == null || imageName.isEmpty()) {
				file = new File("image.png");
			} else {
				file = new File(imageName.concat(".png"));
			}
			ImageIO.write(img, "PNG", file);
		}
		public void printImage() throws IOException {this.printImage("");}
		public int getRows() {return rows;}
		public int getColumns() {return columns;}
}

public class Light {
		private Vector3D lightVector;
		private RGB color;
		public Light(Vector3D vector, RGB color) {
```

```
            this.lightVector = vector;
            this.color = color;
        }
        public Vector3D getLightVector() {return lightVector;}
        public RGB getColor() {return color;}
}

public class OrthonormalBasis {
        Vector3D u;
        Vector3D w;
        Vector3D v;
        final static Vector3D n = new Vector3D(1, 0, 0);
        final static Vector3D m = new Vector3D(0, 1, 0);
        final static double EPSILON = 0.01;
        public OrthonormalBasis() {}
        public OrthonormalBasis(Vector3D u, Vector3D v, Vector3D w) {
            this.u = u;
            this.v = v;
            this.w = w;
        }
        public static OrthonormalBasis constructFromUV(Vector3D a, Vector3D b) {
            Vector3D u = a.makeUnitVector();
            Vector3D w = a.getCrossProduct(b).makeUnitVector();
            Vector3D v = w.getCrossProduct(u);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromVU(Vector3D a, Vector3D b) {
            Vector3D v = a.makeUnitVector();
            Vector3D w = b.getCrossProduct(a).makeUnitVector();
            Vector3D u = v.getCrossProduct(w);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromVW(Vector3D a, Vector3D b) {
            Vector3D v = a.makeUnitVector();
            Vector3D u = a.getCrossProduct(b).makeUnitVector();
            Vector3D w = u.getCrossProduct(v);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromWV(Vector3D a, Vector3D b) {
            Vector3D w = a.makeUnitVector();
            Vector3D u = b.getCrossProduct(a).makeUnitVector();
            Vector3D v = w.getCrossProduct(u);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromUW(Vector3D a, Vector3D b) {
            Vector3D u = a.makeUnitVector();
            Vector3D v = b.getCrossProduct(a).makeUnitVector();
            Vector3D w = u.getCrossProduct(v);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromWU(Vector3D a, Vector3D b) {
            Vector3D w = a.makeUnitVector();
            Vector3D v = a.getCrossProduct(b).makeUnitVector();
            Vector3D u = v.getCrossProduct(w);
            return new OrthonormalBasis(u, v, w);
        }
        public static OrthonormalBasis constructFromU(Vector3D a) {
            Vector3D u = a.makeUnitVector();
            Vector3D v = u.getCrossProduct(n);
            if (v.getLengthSquared() < EPSILON) {
                v = u.getCrossProduct(m);
```

```
                }
                Vector3D w = u.getCrossProduct(v);
                return new OrthonormalBasis(u, v, w);
        }

        public OrthonormalBasis constructFromV(Vector3D a) {
                Vector3D v = a.makeUnitVector();
                Vector3D u = v.getCrossProduct(n);
                if (u.getLengthSquared() < EPSILON) {
                        u = v.getCrossProduct(m);
                }
                Vector3D w = u.getCrossProduct(v);
                return new OrthonormalBasis(u, v, w);
        }
        public OrthonormalBasis constructFromW(Vector3D a) {
                Vector3D w = a.makeUnitVector();
                Vector3D u = w.getCrossProduct(n);
                if (u.getLengthSquared() < EPSILON) {
                        u = w.getCrossProduct(m);
                }
                Vector3D v = w.getCrossProduct(u);
                return new OrthonormalBasis(u, v, w);
        }
        public Vector3D getU() {return u;}
        public void setU(Vector3D u) {this.u = u;}
        public Vector3D getW() {return w;}
        public void setW(Vector3D w) {this.w = w;}
        public Vector3D getV() {return v;}
        public void setV(Vector3D v) {this.v = v;}
}

public class Ray {
        Vector3D origin;
        Vector3D distanceVector;
        public Ray(Vector3D origin, Vector3D vector) {
                this.origin = origin;
                this.distanceVector = vector;
        }
        public Vector pointAtParameter(double t) {
                return origin.add(distanceVector.scaleUp(t));}

        @Override
        public String toString() {
                return "Ray [origin=" + origin + ", distanceVector=" +
distanceVector+ "]";}
        public Vector3D getOrigin() {return origin;}

        public Vector3D getDistanceVector() {return distanceVector;}

}

public class RGB {
        public static final int MAX_BYTE = 255;
        public static final int BYTE_MULTIPLICATION = 256;
        public static final double MAC_GAMMA = 1.8;
        public static final double WINDOWS_GAMMA = 2.2;
        public int red;
        public int green;
        public int blue;
        public RGB(int red, int green, int blue) {
                this.red = red;
```

```
		this.green = green;
		this.blue = blue;
	}
	public void clamp() {
		if (red > 255)
			red = 255;
		if (green > 255)
			green = 255;
		if (blue > 255)
			blue = 255;
		if (red < 0)
			red = 0;
		if (green < 0)
			green = 0;
		if (blue < 0)
			blue = 0;
	}
	public void makePositive() {
		this.setRed(Math.abs(this.red));
		this.setGreen((this.green));
		this.setRed(Math.abs(this.blue));
	}
	public void makeNegative() {
		this.setRed(this.red * -1);
		this.setGreen(this.green * -1);
		this.setBlue(this.blue * -1);
	}
	public RGB multiplyByScalar(float scale) {
		RGB newColor = new RGB(determineScaled(this.red, scale),
determineScaled(this.green, scale),
				determineScaled(this.blue, scale));
		newColor.clamp();
		return newColor;
	}
	private int determineScaled(int color, float scale) {
		int scaled = (int) (color * scale);
		if (scaled <= MAX_BYTE) {
			return scaled;
		} else {
			return scaled / 255;
		}
	}
	public RGB add(RGB b) {
		int red = this.red + b.red;
		int green = this.green + b.green;
		int blue = this.blue + b.blue;
		RGB newColor = new RGB(red, green, blue);
		newColor.clamp();
		return newColor;
	}

	public RGB subtract(RGB b) {
		int red = this.red - b.red;
		int green = this.green - b.green;
		int blue = this.blue - b.blue;
		return new RGB(red, green, blue);
	}
	public RGB multiply(RGB b) {
		int red = (this.red * b.red) / 255;
		int green = (this.green * b.green) / 255;
		int blue = (this.blue * b.blue) / 255;
```

```
			RGB newColor = new RGB(red, green, blue);
			newColor.clamp();
			return newColor;
		}
		public RGB divide(RGB b) {
			int red = this.red / b.red;
			int green = this.green / b.green;
			int blue = this.blue / b.blue;
			RGB newColor = new RGB(red, green, blue);
			newColor.clamp();
			return newColor;
		}
		public RGB getByteForm() {
			int red = colorToByte(this.red);
			int green = colorToByte(this.green);
			int blue = colorToByte(this.blue);
			return new RGB(red, green, blue);
		}
		private int colorToByte(int color) {
			int i = (int) (BYTE_MULTIPLICATION * Math.pow(color, 1 /
MAC_GAMMA));
			if (i > MAX_BYTE) {
				return MAX_BYTE;
			} else {
				return i;
			}
		}
		public int getRed() {return red;}
		public void setRed(int red) {this.red = red;}
		public int getGreen() {return green;}
		public void setGreen(int green) {this.green = green;}
		public int getBlue() {return blue;}
		public void setBlue(int blue) {this.blue = blue;}
		public String toString() {
			return String.format("[%d, %d, %d]", red, green, blue);}
}

public class TransformMatrix {
	Matrix matrix;
	Matrix inverse;
	Matrix identity = Matrix
			.constructWithCopy(Matrix.identity(4, 4).getArray());
	public TransformMatrix(double[] scalars) {
		double[][] arrayMatrix = new double[4][4];
		for (int i = 0; i < 3; i++) {
			for (int j = 0; j < 4; j++) {
				arrayMatrix[i][j] = scalars[i + j];
			}
		}
		arrayMatrix[3][0] = arrayMatrix[3][1] = arrayMatrix[3][2] = 0;
		arrayMatrix[3][3] = 1;

		matrix = new Matrix(new double[][] { arrayMatrix[0],
arrayMatrix[1],
				arrayMatrix[2], arrayMatrix[3] });
		inverse = matrix.inverse();
	}
	/**
	 * Constructs a transformation matrix to translate by a vector
	 *
	 * @param translation
```

```
 */
public TransformMatrix(Vector3D translation) {
	identity.set(0, 3, translation.getX());
	identity.set(1, 3, translation.getY());
	identity.set(2, 3, translation.getZ());
	this.matrix = identity;
}
/**
 * Constructs a transformation matrix to scale
 *
 * @param scaleX
 * @param scaleY
 * @param scaleZ
 */
public TransformMatrix(double scaleX, double scaleY, double scaleZ) {
	identity.set(0, 0, scaleX);
	identity.set(1, 1, scaleY);
	identity.set(2, 2, scaleZ);
	this.matrix = identity;
}
public TransformMatrix() {// Used to get identity and rotations}
public Matrix getIdentity() {return Matrix.identity(4, 4);}
public Matrix rotateXBy(double theta) {
	double cosine = Math.cos(theta);
	double sine = Math.sin(theta);
	identity.set(1, 1, cosine);
	identity.set(1, 2, sine * -1);
	identity.set(2, 1, sine);
	identity.set(2, 2, cosine);
	return identity;
}
public Matrix rotateYBy(double theta) {
	double cosine = Math.cos(theta);
	double sine = Math.sin(theta);
	identity.set(0, 0, cosine);
	identity.set(2, 0, sine * -1);
	identity.set(0, 2, sine);
	identity.set(2, 2, cosine);
	return identity;
}
public Matrix rotateZBy(double theta) {
	double cosine = Math.cos(theta);
	double sine = Math.sin(theta);
	identity.set(0, 0, cosine);
	identity.set(0, 1, sine * -1);
	identity.set(1, 0, sine);
	identity.set(1, 1, cosine);
	return identity;
}
public Vector3D transformMatrixAsLocation(double[] point) {
	Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
			{ point[1] }, { point[2] }, { 1 } });
	Matrix result = matrix.times(pointMatrix);
	double[][] array = result.getArray();
	double scale = array[3][0];
	float x = (float) (result.get(0, 0) / scale);
	float y = (float) (result.get(1, 0) / scale);
	float z = (float) (result.get(2, 0) / scale);
	return new Vector3D(x, y, z);
}
public Vector3D transformMatrixAsOffset(Vector3D vector) {
```

```
            Matrix vectorMatrix = new Matrix(new double[][] { { vector.getX()
},
                        { vector.getY() }, { vector.getZ() }, { 0 } });
            Matrix result = matrix.times(vectorMatrix);
            float x = (float) result.get(0, 0);
            float y = (float) result.get(1, 0);
            float z = (float) result.get(2, 0);
            return new Vector3D(x, y, z);
      }
      public Vector3D transformMatrixAsNormal(Vector3D vector) {
            Matrix vectorMatrix = new Matrix(new double[][] { { vector.getX()
},
                        { vector.getY() }, { vector.getZ() }, { 0 } });
            Matrix result = inverse.transpose().times(vectorMatrix);
            float x = (float) result.get(0, 0);
            float y = (float) result.get(1, 0);
            float z = (float) result.get(2, 0);
            return new Vector3D(x, y, z);
      }
      public Vector3D transformInverseAsLocation(double[] point) {
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix result = inverse.times(pointMatrix);
            double[][] array = result.getArray();
            double scale = array[3][0];
            float x = (float) (result.get(0, 0) / scale);
            float y = (float) (result.get(1, 0) / scale);
            float z = (float) (result.get(2, 0) / scale);
            return new Vector3D(x, y, z);
      }
      public Vector3D transformInverseAsOffset(Vector3D vector) {
            Matrix vectorMatrix = new Matrix(new double[][] { { vector.getX()
},
                        { vector.getY() }, { vector.getZ() }, { 0 } });
            Matrix result = inverse.times(vectorMatrix);
            float x = (float) result.get(0, 0);
            float y = (float) result.get(1, 0);
            float z = (float) result.get(2, 0);
            return new Vector3D(x, y, z);
      }
      public Vector3D transformInverseAsNormal(Vector3D vector) {
            Matrix vectorMatrix = new Matrix(new double[][] { { vector.getX()
},
                        { vector.getY() }, { vector.getZ() }, { 0 } });
            Matrix result = matrix.transpose().times(vectorMatrix);
            float x = (float) result.get(0, 0);
            float y = (float) result.get(1, 0);
            float z = (float) result.get(2, 0);
            return new Vector3D(x, y, z);
      }
      public Matrix rotateUVWToXYZ(OrthonormalBasis uvw) {
            Vector3D u = uvw.getU();
            Vector3D v = uvw.getV();
            Vector3D w = uvw.getW();
            return new Matrix(new double[][] { { u.getX(), u.getY(), u.getZ(),
0 },
                        { v.getX(), v.getY(), v.getZ(), 0 },
                        { w.getX(), w.getY(), w.getZ(), 0 }, { 0, 0, 0, 1 }
});
      }
      public Matrix rotateXYZToUVW(OrthonormalBasis uvw) {
```

```
            Vector3D u = uvw.getU();
            Vector3D v = uvw.getV();
            Vector3D w = uvw.getW();
            return new Matrix(new double[][] { { u.getX(), v.getX(), w.getX(),
0 },
                        { u.getY(), v.getY(), w.getY(), 0 },
                        { u.getZ(), v.getZ(), w.getZ(), 0 }, { 0, 0, 0, 1 }
});
      }
      public double[] moveXYZPointToNewOrigin(double[] point, double[]
newOrigin) {
            identity.set(0, 3, newOrigin[0] * -1);
            identity.set(1, 3, newOrigin[1] * -1);
            identity.set(2, 3, newOrigin[2] * -1);
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix newPoint = identity.times(pointMatrix);
            return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                        newPoint.get(2, 0) };
      }
      public double[] moveXYZPointBackToOriginalOrigin(double[] point,
                  double[] newOrigin) {
            identity.set(0, 3, newOrigin[0]);
            identity.set(1, 3, newOrigin[1]);
            identity.set(2, 3, newOrigin[2]);
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix newPoint = identity.times(pointMatrix);
            return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                        newPoint.get(2, 0) };
      }
      public double[] getXYZinUVWCoordinate(OrthonormalBasis uvw, double[]
point) {
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix newPoint = rotateXYZToUVW(uvw).times(pointMatrix);
            return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                        newPoint.get(2, 0) };
      }
      public double[] getUVWinXYZCoordinate(OrthonormalBasis uvw, double[]
point) {
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix newPoint = rotateUVWToXYZ(uvw).times(pointMatrix);
            return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                        newPoint.get(2, 0) };
      }
      public double[] moveUVWToNewOrigin(OrthonormalBasis uvw, double[] point,
                  double[] newOrigin) {
            identity.set(0, 3, newOrigin[0]);
            identity.set(1, 3, newOrigin[1]);
            identity.set(2, 3, newOrigin[2]);
            Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                        { point[1] }, { point[2] }, { 1 } });
            Matrix xyz = rotateXYZToUVW(uvw).times(pointMatrix);
            Matrix newPoint = identity.times(xyz);
            return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                        newPoint.get(2, 0) };
      }
      public double[] moveUVWBackToOrigin(OrthonormalBasis uvw, double[] point,
                  double[] newOrigin) {
```

```
                identity.set(0, 3, newOrigin[0] * -1);
                identity.set(1, 3, newOrigin[1] * -1);
                identity.set(2, 3, newOrigin[2] * -1);
                Matrix pointMatrix = new Matrix(new double[][] { { point[0] },
                                { point[1] }, { point[2] }, { 1 } });
                Matrix newPoint = rotateUVWToXYZ(uvw)
                                .times(identity.times(pointMatrix));
                return new double[] { newPoint.get(0, 0), newPoint.get(1, 0),
                                newPoint.get(2, 0) };
        }
}

public interface Vector {
        public float getX();
        public float getY();
        public void setX(float x);
        public void setY(float y);
        public double getLength();
        public double getLengthSquared();
        public void makePositive();
        public void makeNegative();
}

public class Vector2D implements Vector {
        float[] vector = new float[2];
        public Vector2D(float x, float y) {
                vector[0] = x;
                vector[1] = y;
        }
        public float getX() {return vector[0];}
        public float getY() {return vector[1];}
        public void setX(float x) {vector[0] = x;}
        public void setY(float y) {vector[1] = y;}
        public double getLength() {
                return Math.sqrt(this.getX() * this.getX() + this.getY() *
this.getY());}
        public double getLengthSquared() {return Math.pow(getLength(), 2);}
        public void makePositive() {
                this.setX(Math.abs(getX()));
                this.setY((getY()));
        }
        public void makeNegative() {
                this.setX(getX() * -1);
                this.setY(getY() * -1);
        }
        public Vector2D scaleUp(double scale) {
                float x = (float) (getX() * scale);
                float y = (float) (getY() * scale);
                return new Vector2D(x, y);
        }
        public Vector2D scaleDown(double scale) {
                float x = (float) (getX() / scale);
                float y = (float) (getY() / scale);
                return new Vector2D(x, y);
        }
        public Vector2D add(Vector2D b) {
                float x = getX() + b.getX();
                float y = getY() + b.getY();
                return new Vector2D(x, y);
        }
        public Vector2D subtract(Vector2D b) {
```

```
                float x = getX() - b.getX();
                float y = getY() - b.getY();
                return new Vector2D(x, y);
        }
        public Vector2D multiply(Vector2D b) {
                float x = getX() * b.getX();
                float y = getY() * b.getY();
                return new Vector2D(x, y);
        }
        public Vector2D divide(Vector2D b) {
                float x = getX() / b.getX();
                float y = getY() / b.getY();
                return new Vector2D(x, y);
        }
}

public class Vector3D implements Vector {
        float[] vector = new float[3];
        public Vector3D(float x, float y, float z) {
                vector[0] = x;
                vector[1] = y;
                vector[2] = z;
        }
        public float getX() {return vector[0];}
        public float getY() {return vector[1];}
        public float getZ() {return vector[2];}
        public void setX(float x) {vector[0] = x;}
        public void setY(float y) {vector[1] = y;}
        public void setZ(float z) {vector[2] = z;}
        public double getLength() {
                return Math.sqrt(getX() * getX() + getY() * getY() + getZ() *
getZ());}
        public double getLengthSquared() {return getLength() * getLength();}
        public Vector3D makeUnitVector() {
                Vector3D newVector = new Vector3D(this.getX(), this.getY(),
this.getZ());
                return newVector.scaleDown(newVector.getLength());
        }
        public void makePositive() {
                this.setX(Math.abs(getX()));
                this.setY((getY()));
                this.setZ(Math.abs(getZ()));
        }
        public void makeNegative() {
                this.setX(getX() * -1);
                this.setY(getY() * -1);
                this.setZ(getZ() * -1);
        }
        public Vector3D multiply(double scale) {
                float x = (float) (getX() * scale);
                float y = (float) (getY() * scale);
                float z = (float) (getZ() * scale);
                return new Vector3D(x, y, z);
        }
        public Vector3D scaleUp(double scale) {
                float x = (float) (getX() * scale);
                float y = (float) (getY() * scale);
                float z = (float) (getZ() * scale);
                return new Vector3D(x, y, z);
        }
        public Vector3D scaleDown(double scale) {
```

```
			float x = (float) (getX() / scale);
			float y = (float) (getY() / scale);
			float z = (float) (getZ() / scale);
			return new Vector3D(x, y, z);
		}
		public Vector3D add(Vector3D b) {
			float x = getX() + b.getX();
			float y = getY() + b.getY();
			float z = getZ() + b.getZ();
			return new Vector3D(x, y, z);
		}
		public Vector3D subtract(Vector3D b) {
			float x = getX() - b.getX();
			float y = getY() - b.getY();
			float z = getZ() - b.getZ();
			return new Vector3D(x, y, z);
		}
		public Vector3D multiply(Vector3D b) {
			float x = getX() * b.getX();
			float y = getY() * b.getY();
			float z = getZ() * b.getZ();
			return new Vector3D(x, y, z);
		}
		public Vector3D divide(Vector3D b) {
			float x = getX() / b.getX();
			float y = getY() / b.getY();
			float z = getZ() / b.getZ();
			return new Vector3D(x, y, z);
		}
		// Used to compute cosine of angle between two vectors
		public float getDotProduct(Vector3D b) {
			return getX() * b.getX() + getY() * b.getY() + getZ() * b.getZ();
		}
		// Resulting vector is perpendicular to both vectors, can be used to find
		// sine of angle between vectors
		public Vector3D getCrossProduct(Vector3D b) {
			float x = getY() * b.getZ() - getZ() * b.getY();
			float y = getZ() * b.getX() - getX() * b.getZ();
			float z = getX() * b.getY() - getY() * b.getX();
			return new Vector3D(x, y, z);
		}
		@Override
		public String toString() {
			return "Vector3D [vector=" + Arrays.toString(vector) + "]";
		}
}

package raytracer.textures;

public interface Texture {
	public RGB getValue(Vector2D vector2, Vector3D vector3);}

public class ImageTexture implements Texture {
	Image image;
	public ImageTexture(String fileName) {
		try {
			File imgFile = new File(fileName);
			BufferedImage imageFile = ImageIO.read(imgFile);
			this.image = new Image(imageFile.getHeight(),
imageFile.getWidth());
		this.image.populateImage(getPixelsFromImage(imageFile));
```

```
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      public RGB[][] getPixelsFromImage(BufferedImage image) {
            RGB[][] pixels = new RGB[image.getHeight()][image.getWidth()];
            final byte[] originalPixels = ((DataBufferByte)
image.getData().getDataBuffer()).getData();
            int rows = 0;
            int columns = 0;
            for (int pixel = 0; pixel < originalPixels.length; pixel += 3) {
                  int r = ((originalPixels[pixel + 2] & 0xff));
                  int g = ((originalPixels[pixel + 1] & 0xff));
                  int b = (originalPixels[pixel] & 0xff);
                  RGB color = new RGB(r, g, b);
                  color.clamp();
                  pixels[rows][columns] = color;
                  columns++;
                  if (columns == image.getWidth()) {
                        columns = 0;
                        rows++;
                        if (rows == image.getHeight()) {
                              break;
                        }
                  }
            }
            return pixels;
      }
      public RGB getValue(Vector2D uv, Vector3D p) {
            float u = uv.getX() - (int) uv.getX();
            float v = uv.getY() - (int) uv.getY();
            u *= image.getRows() - 3;
            v *= image.getColumns() - 3;
            int iu = (int) u;
            int iv = (int) v;
            float tu = u - iu;
            float tv = v - iv;
            RGB[][] pixels = image.image;
            RGB c1 = pixels[iu][iv].multiplyByScalar(1 -
tu).multiplyByScalar(1 - tv);
            RGB c2 = pixels[iu +
1][iv].multiplyByScalar(tu).multiplyByScalar(1 - tv);
            RGB c3 = pixels[iu][iv + 1].multiplyByScalar(1 -
tu).multiplyByScalar(tv);
            RGB c4 = pixels[iu + 1][iv +
1].multiplyByScalar(tu).multiplyByScalar(tv);
            RGB c = c1.add(c2).add(c3).add(c4);
            return c;
      }
}

public class MarbleTexture implements Texture {
      float frequency;
      float scale;
      int octaves;
      RGB color0, color1, color2;
      SolidNoise noise = new SolidNoise();
      public MarbleTexture(float stripesPerUnit, float scale, int octaves) {
            frequency = (float) (Math.PI * stripesPerUnit);
            this.scale = scale;
            this.octaves = octaves;
```

```
            color0 = new RGB(204, 204, 204);
            color1 = new RGB(102, 51, 26);
            color2 = new RGB(15, 10, 5);
      }
      public MarbleTexture(RGB color0, RGB color1, RGB color2,
                  float stripesPerUnit, float scale, int octaves) {
            frequency = (float) (Math.PI * stripesPerUnit);
            this.scale = scale;
            this.octaves = octaves;
            this.color0 = color0;
            this.color1 = color1;
            this.color2 = color2;
      }
      @Override
      public RGB getValue(Vector2D vector2, Vector3D vector3) {
            float temp = scale
                        * noise.createTurbulence(vector3.multiply(frequency),
octaves);
            float t = (float) (2.0f * Math.abs(Math.sin(frequency *
vector3.getX()
                        + temp)));
            if (t < 1.0f) {
                  return color1.multiplyByScalar(t).add(
                              color2.multiplyByScalar(1.0f - t));
            } else {
                  t -= 1.0f;
                  return color0.multiplyByScalar(t).add(
                              color1.multiplyByScalar(1.0f - t));
            }
      }
}

public class NoiseTexture implements Texture {
      float scale;
      RGB color1;
      RGB color2;
      SolidNoise solidNoise = new SolidNoise();
      public NoiseTexture() {
            scale = 1;
            color1 = new RGB(204, 0, 0);
            color2 = new RGB(0, 0, 204);
      }
      public NoiseTexture(float scale, RGB color1, RGB color2) {
            this.scale = scale;
            this.color1 = color1;
            this.color2 = color2;
      }
      @Override
      public RGB getValue(Vector2D vector2, Vector3D vector3) {
            float t = (1.0f + solidNoise.getNoise(vector3.scaleUp(scale))) /
2.0f;
            return color1.multiplyByScalar(t)
                        .add(color2.multiplyByScalar(1.0f - t));
      }
}

public class SolidNoise {
      private Vector3D[] grad = new Vector3D[16];
      private int[] phi = new int[16];
      public SolidNoise() {
            populateGrad();
```

```
		for (int i = 0; i < 16; i++) {
			phi[i] = i;
		}
		// shuffle phi
		for (int i = 14; i >= 0; i--) {
			int target = (int) (Math.random() * i);
			int temp = phi[i + 1];
			phi[i + 1] = phi[target];
			phi[target] = temp;
		}
	}
	private void populateGrad() {
		grad[0] = new Vector3D(1, 1, 0);
		grad[1] = new Vector3D(-1, 1, 0);
		grad[2] = new Vector3D(1, -1, 0);
		grad[3] = new Vector3D(-1, -1, 0);
		grad[4] = new Vector3D(1, 0, 1);
		grad[5] = new Vector3D(-1, 0, 1);
		grad[6] = new Vector3D(1, 0, -1);
		grad[7] = new Vector3D(-1, 0, -1);
		grad[8] = new Vector3D(0, 1, 1);
		grad[9] = new Vector3D(0, -1, 1);
		grad[10] = new Vector3D(0, 1, -1);
		grad[11] = new Vector3D(0, -1, -1);
		grad[12] = new Vector3D(1, 0, 1);
		grad[13] = new Vector3D(-1, 0, 1);
		grad[14] = new Vector3D(1, 0, -1);
		grad[15] = new Vector3D(-1, 0, -1);
	}
	private float getOmega(float t) {
		if (t < 0.0f) {t = t * -1;}
		float num = (-6.0f * t * t * t * t * t + 15.0f * t * t * t * t -
10.0f
				* t * t * t + 1.0f);
		if (Float.isNaN(num)) {return Float.MIN_VALUE;}
		else {return num;}
	}
	private Vector3D getGamma(int i, int j, int k) {
		int index;
		index = phi[Math.abs(k) % 16];
		index = phi[Math.abs(j + index) % 16];
		index = phi[Math.abs(i + index) % 16];
		return grad[index];
	}
	private float getKnot(int i, int j, int k, Vector3D v) {
		float omegas = getOmega(v.getX()) * getOmega(v.getY())
				* getOmega(v.getZ());
		if (Float.isNaN(omegas) || omegas < Float.MIN_VALUE) {
			omegas = Float.MIN_VALUE;
		} else if (omegas > Float.MAX_VALUE) {
			omegas = 0.0f;
		}
		return omegas * getGamma(i, j, k).getDotProduct(v);
	}
	private int getIntGamma(int i, int j) {
		int index;
		index = phi[Math.abs(j) % 16];
		index = phi[Math.abs(i + index) % 16];
		return index;
	}
	public float createTurbulence(Vector3D p, int depth) {
```

```
		float sum = 0.0f;
		float weight = 1.0f;
		Vector3D temp = p;
		sum = Math.abs(getNoise(temp));
		for (int i = 1; i < depth; i++) {
			weight = weight * 2;
			temp.setX(p.getX() * weight);
			temp.setY(p.getY() * weight);
			temp.setZ(p.getZ() * weight);
			sum += Math.abs(getNoise(temp)) / weight;
		}
		return sum;
	}
	public float getDTurbulence(Vector3D p, int depth, float d) {
		float sum = 0.0f;
		float weight = 1.0f;
		Vector3D temp = p;
		sum += Math.abs(getNoise(temp)) / d;
		for (int i = 1; i < depth; i++) {
			weight = weight * 2;
			temp.setX(p.getX() * weight);
			temp.setY(p.getY() * weight);
			temp.setZ(p.getZ() * weight);
			sum += Math.abs(getNoise(temp)) / weight;
		}
		return sum;
	}
	public float getNoise(Vector3D temp) {
		int fi = (int) Math.floor(temp.getX());
		int fj = (int) Math.floor(temp.getY());
		int fk = (int) Math.floor(temp.getZ());
		float fu = temp.getX() - fi;
		float fv = temp.getY() - fj;
		float fw = temp.getZ() - fk;
		float sum = 0.0f;
		Vector3D vector = new Vector3D(fu, fv, fw);
		sum += getKnot(fi, fj, fk, vector);
		vector = new Vector3D(fu - 1, fv, fw);
		sum += getKnot(fi + 1, fj, fk, vector);
		vector = new Vector3D(fu, fv - 1, fw);
		sum += getKnot(fi, fj + 1, fk, vector);
		vector = new Vector3D(fu, fv, fw - 1);
		sum += getKnot(fi, fj, fk + 1, vector);
		vector = new Vector3D(fu - 1, fv - 1, fw);
		sum += getKnot(fi + 1, fj + 1, fk, vector);
		vector = new Vector3D(fu - 1, fv, fw - 1);
		sum += getKnot(fi + 1, fj, fk + 1, vector);
		vector = new Vector3D(fu, fv - 1, fw - 1);
		sum += getKnot(fi, fj + 1, fk + 1, vector);
		vector = new Vector3D(fu - 1, fv - 1, fw - 1);
		sum += getKnot(fi + 1, fj + 1, fk + 1, vector);
		return sum;
	}
}
```

```
package raytracer.shapes;

public interface Surface {
	public double getT();
	public boolean hit(Ray ray, double tSubZero, double tSub1, float time);
	public RGB getColor();
```

```
	public float getReflectance();
	public boolean shadowHit(Ray ray, float tSubZero, float tSub1, float
time);
	public RGB getAmbientColor(float ambience, Vector3D hitPoint);
	public Texture getTexture();
}

public class SurfaceList implements Surface {
	List<Surface> surfaces;
	double t;
	Surface prim;
	public SurfaceList() {surfaces = new ArrayList<>();}
	public void add(Surface surface) {surfaces.add(surface);}
	@Override
	public boolean hit(Ray ray, double tSubZero, double tSub1, float time) {
		boolean hitOne = false;
		t = tSub1;
		prim = null;
		for (Surface surface : surfaces) {
			if (surface.hit(ray, tSubZero, t, time)) {
				prim = surface;
				t = surface.getT();
				hitOne = true;
			}
		}
		return hitOne;
	}
	@Override
	public boolean shadowHit(Ray ray, float tSubZero, float tSub1, float
time) {return false;}
	public Surface getPrim() {return prim;}
	@Override
	public double getT() {return t;}
	@Override
	public RGB getColor() {return prim.getColor();}
	@Override
	public float getReflectance() {return prim.getReflectance();}
	@Override
	public RGB getAmbientColor(float ambience, Vector3D hitPoint) {
		return prim.getAmbientColor(ambience, hitPoint);}
	@Override
	public Texture getTexture() {return prim.getTexture();}
}

public class Sphere implements Surface {
	Vector3D origin;
	float radius;
	double t;
	RGB color;
	float reflectance;
	Texture texture;
	public Sphere(float x, float y, float z, float radius, RGB color, float
reflectance) {
		this.origin = new Vector3D(x, y, z);
		this.radius = radius;
		this.color = color;
		this.reflectance = reflectance;
	}
	public void addTexture(Texture texture) {
		this.texture = texture;
	}
```

```
	public Vector3D getNormalForPoint(Vector3D point) {
		Vector3D normal = point.subtract(origin);
		return normal.scaleDown(radius);
	}
	public float getNDotL(Vector3D point, Vector3D lightVector) {
		Vector3D normal = getNormalForPoint(point);
		return normal.getDotProduct(lightVector);
	}
	public RGB getLitColor(Light light, Vector3D point, float ambience) {
		float nDotL = getNDotL(point, light.getLightVector());
		if (ambience == 0) {
			RGB multipliedLight =
light.getColor().multiply(getAmbientColor(ambience, point));
			return multipliedLight.multiplyByScalar(nDotL);
		} else {
			int color = (int) ((ambience + (getReflectance() * nDotL))
* 255);
			return new RGB(color, color, color);
		}
	}
	public RGB getAmbientColor(float ambience, Vector3D hitPoint) {
		RGB original = getColor();
		double reflectance = getReflectance();
		if (texture != null) {
			if (texture instanceof ImageTexture) {
				float theta = (float) Math.acos((hitPoint.getZ() -
this.origin.getZ()) / this.radius);
				float phi = (float) Math.atan2(hitPoint.getY() -
this.origin.getY(),
						hitPoint.getX() - this.origin.getX());
				if(phi < 0){
					phi += 2*Math.PI;
				}
				float u = (float) (phi / (2 * Math.PI));
				float v = (float) ((Math.PI - theta) / Math.PI);
				original = texture.getValue(new Vector2D(u, v),
hitPoint);
			} else {
				original = texture.getValue(null, hitPoint);
			}
		}
		if (reflectance == 0) {return original;}
		 else {
			return new RGB((int) (original.getRed() * reflectance),
(int) (original.getGreen() * reflectance),
					(int) (original.getBlue() * reflectance));
		}
	}
	@Override
	public boolean hit(Ray ray, double tSubZero, double tSubOne, float time)
{
		Vector3D d = ((Vector3D) ray.getDistanceVector());
		Vector3D originCenter = ray.getOrigin().subtract(origin);
		float a = d.getDotProduct(d);
		float b = 2 * d.getDotProduct(originCenter);
		float c = (float) (originCenter.getDotProduct(originCenter) -
(radius * radius));
		float discriminant = b * b - 4 * a * c;
		if (discriminant > 0) {
			double sqrtd = Math.sqrt(discriminant);
			float tVal = (float) ((-b - sqrtd) / (2 * a));
```

```
                        if (tVal < tSubZero) {
                                tVal = (float) ((-b + sqrtd) / (2 * a));
                        }
                        if (tVal < tSubZero || tVal > tSubOne) {
                                return false;
                        }
                        t = tVal;
                        return true;
                }
                return false;
        }
        @Override
        public boolean shadowHit(Ray ray, float tSubZero, float tSub1, float
time) {
                Vector3D direction = ((Vector3D) ray.getDistanceVector());
                Vector3D temp = ray.getOrigin().subtract(origin);
                float a = direction.getDotProduct(direction);
                float b = 2 * direction.getDotProduct(temp);
                float c = (float) (temp.getDotProduct(temp) - (radius * radius));
                float discriminant = b * b - 4 * a * c;
                if (discriminant > 0) {
                        double sqrtd = Math.sqrt(discriminant);
                        float tVal = (float) ((-b - sqrtd) / (2 * a));
                        if (tVal < tSubZero) {
                                tVal = (float) ((-b + sqrtd) / (2 * a));
                        }
                        if (tVal < tSubZero || tVal > tSub1) {
                                return false;
                        }
                        return true;
                }
                return false;
        }
        public double getT() {return t;}
        public RGB getColor() {return color;}
        public float getReflectance() {return reflectance;}
        @Override
        public Texture getTexture() {return texture;}
}

public class Triangle implements Surface {
        double[] a = new double[3];
        double[] b = new double[3];
        double[] c = new double[3];
        double t;
        double aX;
        double aY;
        double aZ;
        double bX;
        double bY;
        double bZ;
        double cX;
        double cY;
        double cZ;
        Vector3D d;
        Vector3D origin;
        RGB color;
        float reflectance;
        Texture texture;
        public Triangle(double[] a, double[] b, double[] c, RGB color,
                        float reflectance) {
```

```
		this.a = a;
		this.b = b;
		this.c = c;
		this.aX = this.a[0];
		this.aY = this.a[1];
		this.aZ = this.a[2];
		this.bX = this.b[0];
		this.bY = this.b[1];
		this.bZ = this.b[2];
		this.cX = this.c[0];
		this.cY = this.c[1];
		this.cZ = this.c[2];
		this.color = color;
		this.reflectance = reflectance;
	}
	public void addTexture(Texture texture) {this.texture = texture;}
	public Vector3D getNormal() {
		Vector3D p1Minusp0 = new Vector3D((float) (this.aX - this.cX),
				(float) (this.aY - this.cY), (float) (this.aZ -
this.cZ));
		Vector3D p2Minusp0 = new Vector3D((float) (this.bX - this.cX),
				(float) (this.bY - this.cY), (float) (this.bZ -
this.cZ));
		return p1Minusp0.multiply(p2Minusp0);
	}
	public RGB getLitColor(Light light, float ambience) {
		float nDotL = getNormal().getDotProduct(light.getLightVector());
		if (ambience == 0) {
			RGB multipliedLight = light.getColor().multiply(
					getAmbientColor(ambience, new
Vector3D(1,1,1)));
			return multipliedLight.multiplyByScalar(nDotL);
		} else {
			int color = (int) ((ambience + (getReflectance() * nDotL))
* 255);
			return new RGB(color, color, color);
		}
	}
	@Override
	public RGB getAmbientColor(float ambience, Vector3D hitPoint) {
		RGB original = getColor();
		if (ambience == 0) {
			return addReflectance(new int[] { original.getRed(),
					original.getGreen(), original.getBlue() });
		} else {
			float adjustment = ambience * 255;
			int red = (int) (original.getRed() + adjustment);
			int green = (int) (original.getGreen() + adjustment);
			int blue = (int) (original.getBlue() + adjustment);
			return addReflectance(new int[] { red, green, blue });
		}
	}
	private RGB addReflectance(int[] rgb) {
		double reflectance = getReflectance();
		if (reflectance == 0) {
			return new RGB(rgb[0], rgb[1], rgb[2]);
		} else {
			return new RGB((int) (rgb[0] * reflectance),
					(int) (rgb[1] * reflectance), (int) (rgb[2] *
reflectance));
		}
```

```
	}
	@Override
	public boolean hit(Ray ray, double tSubZero, double tSub1, float time) {
		this.d = ((Vector3D) ray.getDistanceVector());
		this.origin = (Vector3D) ray.getOrigin();
		Matrix matrixA = createMatrixA();
		double determinantA = matrixA.det();
		double beta = getBeta(determinantA);
		if (beta <= 0.0 || beta >= 1.0) {
			return false;
		}
		double gamma = getGamma(determinantA);
		double tVal = setT(determinantA);
		if (gamma <= 0.0 || beta + gamma >= 1.0) {
			return false;
		}
		if (tVal >= tSubZero && tVal <= tSub1) {
			this.t = tVal;
			return true;
		}
		return false;
	}
	private Matrix createMatrixA() {
		return new Matrix(
				new double[][] { { aX - bX, aX - cX, d.getX() },
						{ aY - bY, aY - cY, d.getY() },
						{ aZ - bZ, aZ - cZ, d.getZ() } });
	}
	private double getBeta(double determinantA) {
		Matrix betaNum = createMatrixBeta();
		return betaNum.det() / determinantA;
	}
	private Matrix createMatrixBeta() {
		return new Matrix(new double[][] {
				{ aX - this.origin.getX(), aX - cX, d.getX() },
				{ aY - this.origin.getY(), aY - cY, d.getY() },
				{ aZ - this.origin.getZ(), aZ - cZ, d.getZ() } });
	}
	private double getGamma(double determinantA) {
		Matrix gammaNum = createMatrixGamma();
		return gammaNum.det() / determinantA;
	}
	private Matrix createMatrixGamma() {
		return new Matrix(new double[][] {
				{ aX - bX, aX - this.origin.getX(), d.getX() },
				{ aY - bY, aY - this.origin.getY(), d.getY() },
				{ aZ - bZ, aZ - this.origin.getZ(), d.getZ() } });
	}
	private double setT(double determinantA) {
		Matrix tNum = createMatrixT();
		return tNum.det() / determinantA;
	}

	private Matrix createMatrixT() {
		return new Matrix(new double[][] {
				{ aX - bX, aX - cX, aX - this.origin.getX() },
				{ aY - bY, aY - cY, aY - this.origin.getY() },
				{ aZ - bZ, aZ - cZ, aZ - this.origin.getZ() } });
	}
	public boolean shadowHit(Ray ray, float tSubZero, float tSub1, float
time) {
```

```
            this.d = ((Vector3D) ray.getDistanceVector());
            this.origin = (Vector3D) ray.getOrigin();
            Matrix matrixA = createMatrixA();
            double determinantA = matrixA.det();
            double beta = getBeta(determinantA);
            if (beta <= 0.0 || beta >= 1.0) {return false;}
            double gamma = getGamma(determinantA);
            double tVal = setT(determinantA);
            if (gamma <= 0.0 || beta + gamma >= 1.0) {return false;}
            return (tVal >= tSubZero && tVal <= tSub1);
        }
        @Override
        public double getT() {return t;}
        public RGB getColor() {return color;}
        public float getReflectance() {return this.reflectance;}
        @Override
        public Texture getTexture() {return this.texture;}
}


package raytracerapp;

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}

public class MainActivityFragment extends Fragment {
    PictureCellAdapter adapter;
    ArrayList<PictureCell> cells;
    public MainActivityFragment() { }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View view =  inflater.inflate(R.layout.fragment_main, container,
false);
        ListView list = (ListView) view.findViewById(R.id.listView);
        cells = new ArrayList<PictureCell>();
        // obj tests
        cells.add(new PictureCell(R.drawable.otoneobject, "One Object"));
        cells.add(new PictureCell(R.drawable.otthreeobject, "Three Objects"));
        cells.add(new PictureCell(R.drawable.otelevenobject, "Eleven
Objects"));
        cells.add(new PictureCell(R.drawable.otsixteenobject, "Sixteen
Objects"));
        cells.add(new PictureCell(R.drawable.ottwentyoneobject, "Twenty One
Objects"));
        cells.add(new PictureCell(R.drawable.otsnowimage, "Snowman"));
        //size tests
        cells.add(new PictureCell(R.drawable.stsmallimage, "Small Image"));
        cells.add(new PictureCell(R.drawable.stsmallimage, "Medium Image"));
        cells.add(new PictureCell(R.drawable.stsmallimage, "Large Image"));
        // surface tests
        cells.add(new PictureCell(R.drawable.sutsphereimage, "Sphere Image"));
        cells.add(new PictureCell(R.drawable.suttriangleimage, "Triangle
Image"));
        cells.add(new PictureCell(R.drawable.suttwosphereimage, "Two Sphere
Image"));
```

```
        cells.add(new PictureCell(R.drawable.suttwotriangleimage, "Two Triangle
Image"));
        // color tests
        cells.add(new PictureCell(R.drawable.ctblackandwhite, "Black and White
Image"));
        cells.add(new PictureCell(R.drawable.ctcolored, "Colored Image"));
        cells.add(new PictureCell(R.drawable.ctmarble, "Marble Image"));
        cells.add(new PictureCell(R.drawable.ctnoise, "Noise Image"));
        // light tests
        cells.add(new PictureCell(R.drawable.ltnolightimage, "No Light
Image"));
        cells.add(new PictureCell(R.drawable.ltonelightimage, "Light Image"));
        // texture tests
        cells.add(new PictureCell(R.drawable.ttlargeimage, "Large Texture
Image"));
        cells.add(new PictureCell(R.drawable.ttmediumimage, "Medium Texture
Image"));
        cells.add(new PictureCell(R.drawable.ttsmallimage, "Small Texture
Image"));
        cells.add(new PictureCell(R.drawable.ttxsmallimage, "X-Small Texture
Image"));
        adapter = new PictureCellAdapter(this.getActivity(), cells);
        list.setAdapter(adapter);
        list.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int
position, long id) {
                PictureCell cell = cells.get(position);
                GenericImage img = null;
                switch (position) {
                    case 0: img = new
OneObjectImage(getOutputMediaFile(cell.getFileName())); break;
                    case 1: img = new
ThreeObjectImage(getOutputMediaFile(cell.getFileName()));break;
                    case 2: img = new
ElevenObjectImage(getOutputMediaFile(cell.getFileName()));break;
                    case 3: img = new
SixteenObjectImage(getOutputMediaFile(cell.getFileName()));break;
                    case 4: img = new
TwentyOneObjectImage(getOutputMediaFile(cell.getFileName()));break;
                    case 5: img = new
SnowImage(getOutputMediaFile(cell.getFileName()));break;
                    case 6: img = new
SmallImage(getOutputMediaFile(cell.getFileName()));break;
                    case 7: img = new
MediumImage(getOutputMediaFile(cell.getFileName())); break;
                    case 8: img = new
LargeImage(getOutputMediaFile(cell.getFileName()));break;
                    case 9: img = new
SphereImage(getOutputMediaFile(cell.getFileName()));break;
                    case 10: img = new
TriangleImage(getOutputMediaFile(cell.getFileName()));break;
                    case 11: img = new
TwoSphereImage(getOutputMediaFile(cell.getFileName()));break;
                    case 12: img = new
TwoTriangleImage(getOutputMediaFile(cell.getFileName())); break;
                    case 13: img = new
BlackAndWhiteImage(getOutputMediaFile(cell.getFileName())); break;
                    case 14: img = new
ColorImage(getOutputMediaFile(cell.getFileName()));break;
                    case 15: img = new
```

```
MarbleTextureImage(getOutputMediaFile(cell.getFileName())); break;
                    case 16: img = new
NoiseImage(getOutputMediaFile(cell.getFileName())); break;
                    case 17: img = new
NoLightImage(getOutputMediaFile(cell.getFileName())); break;
                    case 18: img = new
LightImage(getOutputMediaFile(cell.getFileName()));break;
                    case 19:img = new
LargeTextureImage(getActivity().getApplicationContext().getResources(),
getOutputMediaFile(cell.getFileName()), getOutputMediaFile("lgtexture.png"));
break;
                    case 20: img = new
MediumTextureImage(getActivity().getApplicationContext().getResources(),
getOutputMediaFile(cell.getFileName()),
getOutputMediaFile("mdtexture.png"));break;
                    case 21: img = new
SmallTextureImage(getActivity().getApplicationContext().getResources(),
getOutputMediaFile(cell.getFileName()),
getOutputMediaFile("smtexture.png"));break;
                    case 22:img = new
XSmallTextureImage(getActivity().getApplicationContext().getResources(),
getOutputMediaFile(cell.getFileName()), getOutputMediaFile("xsmtexture.png"));
break;
                }
                generateInfo(img, view, cell);
            }
        });
        return view;
    }
    private void generateInfo(GenericImage image, View view, PictureCell cell){
        String time = image.getTime() + " ms.";
        Toast.makeText(view.getContext(),
                "Generating " + cell.getName() + " took " + time,
                Toast.LENGTH_LONG).show();
        cell.setPath(image.getPath());
        cell.setTime(time);
        adapter.notifyDataSetChanged();
    }
    private File getOutputMediaFile(String name) {
        File mediaStorageDirectory = new
File(Environment.getExternalStorageDirectory() +
                "/Android/data/" +
getActivity().getApplicationContext().getPackageName() + "/Files");
        if(!mediaStorageDirectory.exists()){
            mediaStorageDirectory.mkdirs();
        }
        return new File(mediaStorageDirectory.getPath() + File.separator +
name);
    }
}

public class PictureCell {
    public PictureCell(int image, String name) {
        this.image = image;
        this.name = name;
    }
    public int getImage() {return image;}
    public void setImage(int image) {this.image = image;}
    public String getName() { return name;}
    public String getFileName(){return this.name.replace(" ", "_");}
    public void setName(String name) {this.name = name;}
```

```
    private int image;
    private String name;
    private File path;
    private String time;
    public File getPath() { return path;}
    public void setPath(File path) {this.path = path;}
    public String getTime() {return time;}
    public void setTime(String time) {this.time = time;}
}

public class PictureCellAdapter extends ArrayAdapter<PictureCell> {
    private final Context context;
    private final ArrayList<PictureCell> cells;
    public PictureCellAdapter(Context context, ArrayList<PictureCell> cells){
        super(context, R.layout.picture_cell, cells);
        this.context = context;
        this.cells = cells;
    }
    @Override
    public PictureCell getItem(int position) { return cells.get(position);}
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = (LayoutInflater) context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        View cell = inflater.inflate(R.layout.picture_cell, parent, false);
        ImageView image = (ImageView) cell.findViewById(R.id.imageView);
        final PictureCell currentCell = cells.get(position);
        image.setImageResource(currentCell.getImage());
        TextView text = (TextView) cell.findViewById(R.id.name);
        text.setText(currentCell.getName());
        final TextView time = (TextView) cell.findViewById(R.id.time);
        if(currentCell.getTime() != null){
time.setText(currentCell.getTime());}
else {time.setVisibility(View.INVISIBLE); }
        final Button getFile = (Button) cell.findViewById(R.id.file_open);
        final Button redo = (Button) cell.findViewById(R.id.button);
        if(currentCell.getPath() == null){
            getFile.setVisibility(View.INVISIBLE);
            redo.setVisibility(View.INVISIBLE);
        } else {
            getFile.setVisibility(View.VISIBLE);
            redo.setVisibility(View.VISIBLE);
        }
        getFile.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(v.getContext(),
GeneratedImage.class);
                intent.putExtra("PATH",
currentCell.getPath().getAbsolutePath());
                v.getContext().startActivity(intent);
            }
        });
        redo.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                time.setVisibility(View.INVISIBLE);
                getFile.setVisibility(View.INVISIBLE);
                redo.setVisibility(View.INVISIBLE);
            }
        });
```

```
        return cell;
    }
}

package raytracer.utilities;
public class Image {
        int rows;
        int columns;
        public RGB[][] image;
        private SurfaceList surfaces = new SurfaceList();
        private Light light;
        private Camera camera;
        private float ambience;
        private File path;
        public Image(int rows, int columns, File path) {
            this.rows = rows;
            this.columns = columns;
            this.image = new RGB[columns][rows];
            this.path = path;
        }
        public void addSurface(Surface surface) { surfaces.add(surface);}
        public void addLight(Light light) {this.light = light;}
        public void addCamera(Camera camera) {this.camera = camera;}
        public void addAmbientLight(float ambience) {this.ambience = ambience;}
        public void createImage() {
            RGB[][] pixels = new RGB[this.columns][this.rows];
            for (int i = 0; i < this.rows; i++) {
                for (int j = 0; j < this.columns; j++) {
                    Ray ray = this.createRay(i, j);
                    if (this.surfaces.hit(ray, 0, Integer.MAX_VALUE, 0)) {
                        pixels[i][j] = getHitColor(ray);
                    } else {
                        pixels[i][j] = getAmbientBlack();
                    }
                }
            }
            populateImage(pixels);
        }
        private RGB getAmbientBlack() {
            int ambientBlack = (int) (0 + (this.ambience * 255));
            return new RGB(ambientBlack, ambientBlack, ambientBlack);
        }
        private Ray createRay(int i, int j) {
            if (this.camera == null) {
                Vector3D origin = new Vector3D(i, j, 0);
                return new Ray(origin, new Vector3D(0, 0, -1));
            } else {
                return this.camera.getRay(i, j, rows, columns);
            }
        }
        public RGB getHitColor(Ray ray) {
            Surface hitSurface = this.surfaces.getPrim();
            Vector3D hitPoint = (Vector3D) ray.pointAtParameter(this.surfaces
                    .getT());
            if (this.light != null) {
                if (isHitByShadowRay(ray, hitSurface)) {return
getAmbientBlack();} else {
                    if (hitSurface instanceof Sphere) {
                        Sphere sphere = (Sphere) hitSurface;
                        return sphere.getLitColor(light, hitPoint, ambience);
                    } else if (hitSurface instanceof Triangle) {
```

```
                    Triangle tri = (Triangle) hitSurface;
                    return tri.getLitColor(light, ambience);
                } else {
                    return getAmbientBlack();
                }
            }
        } else {return hitSurface.getAmbientColor(ambience, hitPoint);}
    }
    private boolean isHitByShadowRay(Ray ray, Surface hitSurface) {
        Vector3D originOfShadowRay = (Vector3D)
ray.pointAtParameter(hitSurface
                .getT());
        Ray shadowRay = new Ray(originOfShadowRay, light.getLightVector());
        return this.surfaces.hit(shadowRay, 0.001, Integer.MAX_VALUE, 0);
    }
    public void populateImage(RGB[][] pixels) {image = pixels;}
    public void printImage() throws IOException {
        Bitmap img = Bitmap.createBitmap(columns, rows,
Bitmap.Config.ARGB_8888);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                RGB rgb = image[i][j];
                int color = android.graphics.Color.rgb(rgb.red, rgb.green,
rgb.blue);
                img.setPixel(j, i, color);
            }
        }
        FileOutputStream fos = new FileOutputStream(path);
        img.compress(Bitmap.CompressFormat.PNG, 90, fos);
    }
    public int getRows() {return rows;}
    public int getColumns() { return columns;}
}
```